一些约定

字符串相关的定义请参考 字符串基础。

字符串下标从 1 开始。

字符串 s 的长度为 n。

" 后缀 i" 代指以第 i 个字符开头的后缀，存储时用 i 代表字符串 s 的后缀 $s[i \dots n]$。

后缀数组是什么?

后缀数组（Suffix Array）主要关系到两个数组：sa 和 rk。

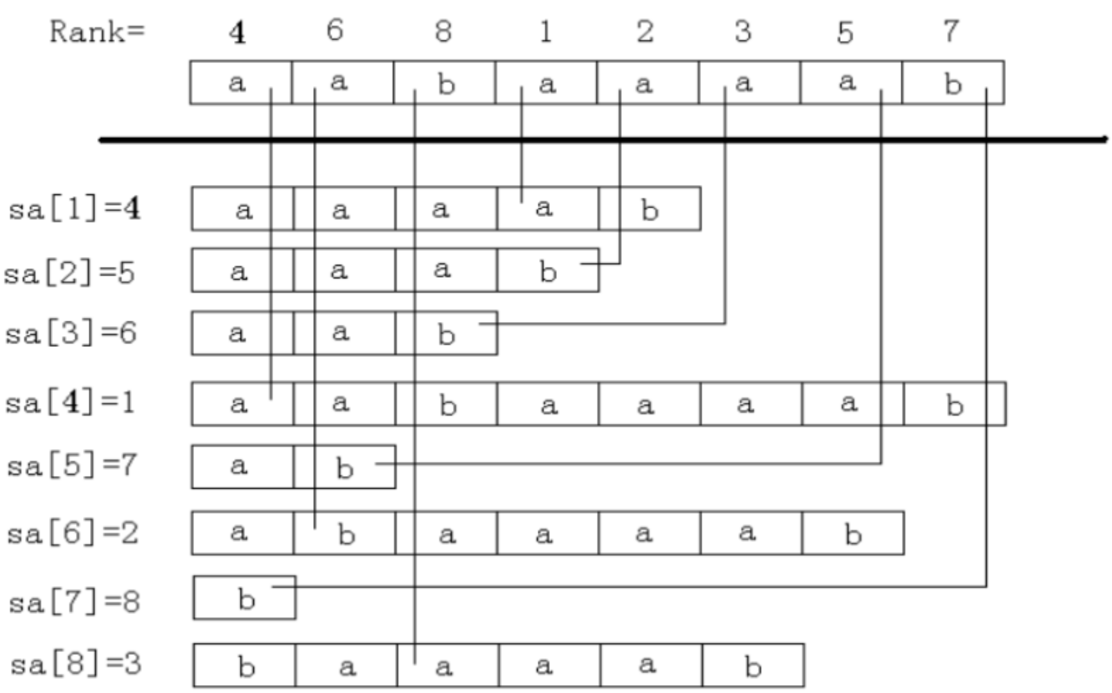
其中，$sa[i]$ 表示将所有后缀排序后第 i 小的后缀的编号，也是所说的后缀数组，后文也称编号数组 sa；

$rk[i]$ 表示后缀 i 的排名，是重要的辅助数组，后文也称排名数组 rk。

这两个数组满足性质：$sa[rk[i]] = rk[sa[i]] = i$。

解释

后缀数组示例：



后缀数组怎么求?

O(n^2logn) 做法

我相信这个做法大家还是能自己想到的：将盛有全部后缀字符串的数组进行 `sort` 排序，由于排序进行 $O(n\log n)$ 次字符串比较，每次字符串比较要 $O(n)$ 次字符比较，所以这个排序是 $O(n^2\log n)$ 的时间复杂度。

O(nlog^2n) 做法

这个做法要用到倍增的思想。

首先对字符串 s 的所有长度为 1 的子串，即每个字符进行排序，得到排序后的编号数组 sa_1 和排名数组 rk_1。

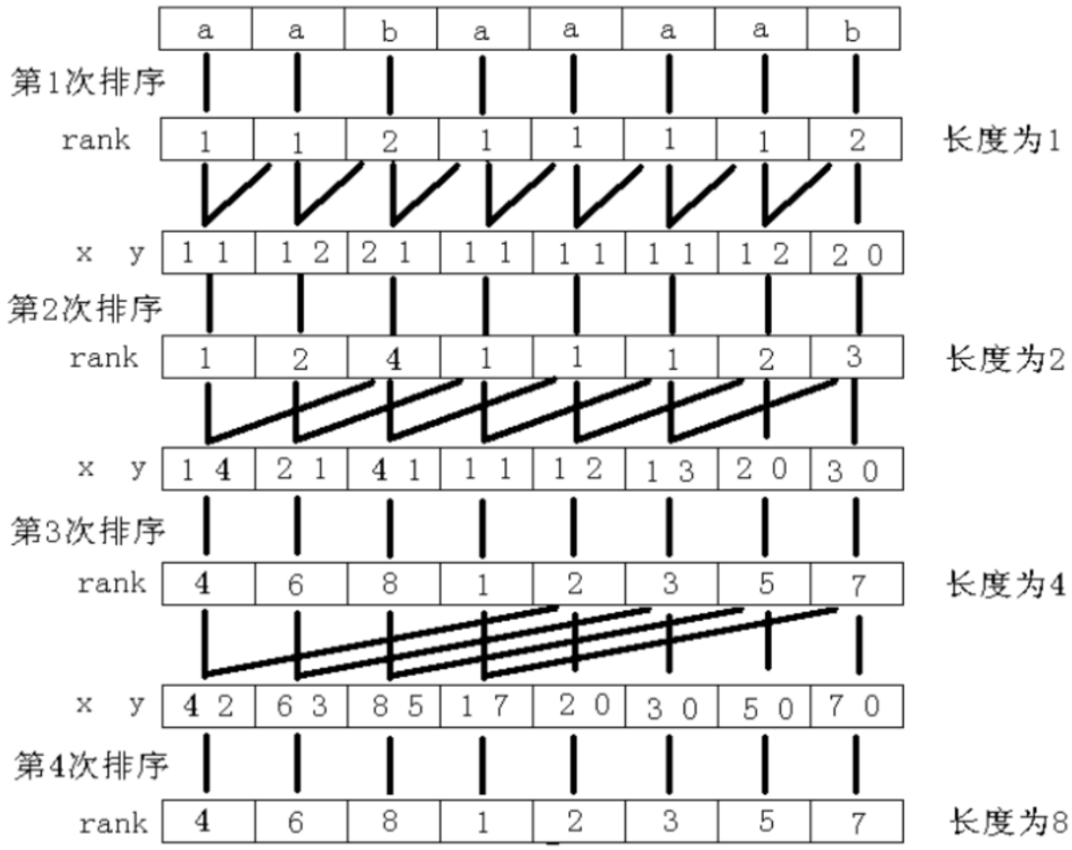
倍增过程：

1. 用两个长度为 1 的子串的排名，即 $rk_1[i]$ 和 $rk_1[i+1]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 2 的子串：$\{s[i\dots min(i+1,n)]\mid i\in[1,\ n]\}$ 进行排序，得到 sa_2 和 rk_2 ；
2. 之后用两个长度为 2 的子串的排名，即 $rk_2[i]$ 和 $rk_2[i+2]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 4 的子串：$\{s[i\dots min(i+3,n)]\mid i\in[1,\ n]\}$ 进行排序，得到 sa_4 和 rk_4；
3. 以此倍增，用长度为 $w/2$ 的子串的排名，即 $rk_{w/2}[i]$ 和 $rk_{w/2}[i+w/2]$，作为排序的第一第二关键字，就可以对字符串 s 的每个长度为 w 的子串 $s[i\dots min(i+w-1,\ n)]$ 进行排序，得到 sa_w 和 rk_w。其中，类似字母序排序规则，当 $i+w>n$ 时，$rk_w[i+w]$ 视为无穷小；
4. $rk_w[i]$ 即是子串 $s[i\dots i+w-1]$ 的排名，这样当 $w\geqslant n$ 时，得到的编号数组 sa_w，也就是我们需要的后缀数组。

过程

倍增排序示意图：

倍增排序示意图：



显然倍增的过程是 $O(\log n)$，而每次倍增用 `sort` 对子串进行排序是 $O(n \log n)$，而每次子串的比较花费 2 次字符比较；

除此之外，每次倍增在 `sort` 排序完后，还有额外的 $O(n)$ 时间复杂度的，更新 rk 的操作，但是相对于 $O(n \log n)$ 被忽略不计；

所以这个算法的时间复杂度就是 $O(n \log^2 n)$。

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;

const int N = 1000010;

char s[N];
int n, w, sa[N], rk[N << 1], oldrk[N << 1];

// 为了防止访问 rk[i+w] 导致数组越界，开两倍数组。
// 当然也可以在访问前判断是否越界，但直接开两倍数组方便一些。

int main() {
  int i, p;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (i = 1; i <= n; ++i) sa[i] = i, rk[i] = s[i];
```

```
  for (w = 1; w < n; w <<= 1) {
    sort(sa + 1, sa + n + 1, [](int x, int y) {
      return rk[x] == rk[y] ? rk[x + w] < rk[y + w] : rk[x] < rk[y];
    });  // 这里用到了 lambda
    memcpy(oldrk, rk, sizeof(rk));
    // 由于计算 rk 的时候原来的 rk 会被覆盖，要先复制一份
    for (p = 0, i = 1; i <= n; ++i) {
      if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
          oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {
        rk[sa[i]] = p;
      } else {
        rk[sa[i]] = ++p;
      }  // 若两个子串相同，它们对应的 rk 也需要相同，所以要去重
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

O(nlogn) 做法

在刚刚的 $O(n\log^2 n)$ 做法中，单次排序是 $O(n\log n)$ 的，如果能 $O(n)$ 排序，就能 $O(n\log n)$ 计算后缀数组了。

前置知识：计数排序，基数排序。

由于计算后缀数组的过程中排序的关键字是排名，值域为 $O(n)$，并且是一个双关键字的排序，可以使用基数排序优化至 $O(n)$。

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;

const int N = 1000010;

char s[N];
int n, sa[N], rk[N << 1], oldrk[N << 1], id[N], cnt[N];

int main() {
  int i, m, p, w;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  m = 127;
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;
  memcpy(oldrk + 1, rk + 1, n * sizeof(int));
  for (p = 0, i = 1; i <= n; ++i) {
    if (oldrk[sa[i]] == oldrk[sa[i - 1]]) {
      rk[sa[i]] = p;
    } else {
      rk[sa[i]] = ++p;
```

```
    }
  }

  m = n;
  for (w = 1; w < n; w <<= 1) {
    // 对第二关键字：id[i] + w进行计数排序
    memset(cnt, 0, sizeof(cnt));
    memcpy(id + 1, sa + 1,
           n * sizeof(int));  // id保存一份儿sa的拷贝，实质上就相当于oldsa
    for (i = 1; i <= n; ++i) ++cnt[rk[id[i] + w]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[rk[id[i] + w]]--] = id[i];

    // 对第一关键字：id[i]进行计数排序
    memset(cnt, 0, sizeof(cnt));
    memcpy(id + 1, sa + 1, n * sizeof(int));
    for (i = 1; i <= n; ++i) ++cnt[rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[rk[id[i]]]--] = id[i];

    memcpy(oldrk + 1, rk + 1, n * sizeof(int));
    for (p = 0, i = 1; i <= n; ++i) {
      if (oldrk[sa[i]] == oldrk[sa[i - 1]] &&
          oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]) {
        rk[sa[i]] = p;
      } else {
        rk[sa[i]] = ++p;
      }
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```

一些常数优化

如果你把上面那份代码交到 LOJ #111: 后缀排序 上：

这是因为，上面那份代码的常数的确很大。

第二关键字无需计数排序

思考一下第二关键字排序的实质，其实就是把超出字符串范围（即 $sa[i] + w > n$）的 $sa[i]$ 放到 sa 数组头部，然后把剩下的依原顺序放入：

```
for (p = 0, i = n; i > n - w; --i) id[++p] = i;

for (i = 1; i <= n; ++i) {
  if (sa[i] > w) id[++p] = sa[i] - w;
}
```

优化计数排序的值域

每次对 rk 进行更新之后，我们都计算了一个 p，这个 p 即是 rk 的值域，将值域改成它即可。

将 rk[id[i]] 存下来，减少不连续内存访问

这个优化在数据范围较大时效果非常明显。

用函数 cmp 来计算是否重复

同样是减少不连续内存访问，在数据范围较大时效果比较明显。

把 `oldrk[sa[i]] == oldrk[sa[i - 1]] && oldrk[sa[i] + w] == oldrk[sa[i - 1] + w]`

替换成 `cmp(sa[i], sa[i - 1], w)`，

`bool cmp(int x, int y, int w) { return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w]; }`。

若排名都不相同可直接生成后缀数组

考虑新的 rk 数组，若其值域为 $[1, n]$ 那么每个排名都不同，此时无需再排序。

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;

const int N = 1000010;

char s[N];
int n, sa[N], rk[N], oldrk[N << 1], id[N], key1[N], cnt[N];
// key1[i] = rk[id[i]]（作为基数排序的第一关键字数组）
int n, sa[N], rk[N], oldrk[N << 1], id[N], px[N], cnt[N];

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, m = 127, p, w;

  scanf("%s", s + 1);
  n = strlen(s + 1);
  for (i = 1; i <= n; ++i) ++cnt[rk[i] = s[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1;; w <<= 1, m = p) {  // m=p 就是优化计数排序值域
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;

    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[key1[i] = rk[id[i]]];
    // 注意这里px[i] != i，因为rk没有更新，是上一轮的排名数组
```

```
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[key1[i]]--] = id[i];
    memcpy(oldrk + 1, rk + 1, n * sizeof(int));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
    if (p == n) {
      for (int i = 1; i <= n; ++i) sa[rk[i]] = i;
      break;
    }
  }

  for (i = 1; i <= n; ++i) printf("%d ", sa[i]);

  return 0;
}
```
```

# 后缀数组的应用

## 寻找最小的循环移动位置

将字符串 $S$ 复制一份变成 $SS$ 就转化成了后缀排序问题。

例题：P4051

### 题目描述

复制Markdown 展开

喜欢钻研问题的JS 同学，最近又迷上了对加密方法的思考。一天，他突然想出了一种他认为是终极的加密办法：把需要加密的信息排成一圈，显然，它们有很多种不同的读法。

例如'JSOI07'，可以读作： JSOI07 SOI07J OI07JS I07JSO 07JSOI 7JSOI0 把它们按照字符串的大小排序：07JSOI 7JSOI0 I07JSO JSOI07 OI07JS SOI07J 读出最后一列字符：I0O7SJ，就是加密后的字符串（其实这个加密手段实在很容易破解，鉴于这是突然想出来的，那就^^）。 但是，如果想加密的字符串实在太长，你能写一个程序完成这个任务吗？

### 输入格式

输入文件包含一行，欲加密的字符串。注意字符串的内容不一定是字母、数字，也可以是符号等。

### 输出格式

输出一行，为加密后的字符串。

### 输入输出样例

**输入** #1 复制

```
JSOI07
```

输出 #1 复制

```
I0O7SJ
```

说明/提示

对于40%的数据字符串的长度不超过10000。

对于100%的数据字符串的长度不超过100000。
```

在字符串中找子串

任务是在线地在主串 T 中寻找模式串 S。在线的意思是，我们已经预先知道知道主串 T，但是当且仅当询问时才知道模式串 S。我们可以先构造出 T 的后缀数组，然后查找子串 S。若子串 S 在 T 中出现，它必定是 T 的一些后缀的前缀。因为我们已经将所有后缀排序了，我们可以通过在 p 数组中二分 S 来实现。比较子串 S 和当前后缀的时间复杂度为 $O(|S|)$，因此找子串的时间复杂度为 $O(|S|\log|T|)$。注意，如果该子串在 T 中出现了多次，每次出现都是在 p 数组中相邻的。因此出现次数可以通过再次二分找到，输出每次出现的位置也很轻松。

从字符串首尾取字符最小化字典序

例题：P2870

> **题目描述**
>
> Farmer John 打算带领 N （$1 \le N \le 5 \times 10^5$）头奶牛参加一年一度的"全美农场主大奖赛"。在这场比赛中，每个参赛者必须让他的奶牛排成一列，然后带领这些奶牛从裁判面前依此走过。
>
> 今年，竞赛委员会在接受报名时，采用了一种新的登记规则：取每头奶牛名字的首字母，按照它们在队伍中的次序排成一列。将所有队伍的名字按字典序升序排序，从而得到出场顺序。
>
> FJ 由于事务繁忙，他希望能够尽早出场。因此他决定重排队列。
>
> 他的调整方式是这样的：每次，他从原队列的首端或尾端牵出一头奶牛，将她安排到新队列尾部。重复这一操作直到所有奶牛都插入新队列为止。
>
> 现在请你帮 FJ 算出按照上面这种方法能排出的字典序最小的队列。

题意：给你一个字符串，每次从首或尾取一个字符组成字符串，问所有能够组成的字符串中字典序最小的一个。
暴力做法就是每次最坏 $O(n)$ 地判断当前应该取首还是尾（即比较取首得到的字符串与取尾得到的反串的大小），只需优化这一判断过程即可。

优化: 由于需要在原串后缀与反串后缀构成的集合内比较大小，可以将反串拼接在原串后，并在中间加上一个没出现过的字符（如 `#`，代码中可以直接使用空字符），求后缀数组，即可 $O(1)$ 完成这一判断。

```
#include <cctype>
#include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;

const int N = 1000010;

char s[N];
int n, sa[N], id[N], oldrk[N << 1], rk[N << 1], key1[N], cnt[N];

bool cmp(int x, int y, int w) {
  return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
  int i, w, m = 127, p, l = 1, r, tot = 0;

  cin >> n;
```

```
  r = n;

  for (i = 1; i <= n; ++i)
    while (!isalpha(s[i] = getchar()))
      ;
  for (i = 1; i <= n; ++i) rk[i] = rk[2 * n + 2 - i] = s[i];

  n = 2 * n + 1;

  for (i = 1; i <= n; ++i) ++cnt[rk[i]];
  for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
  for (i = n; i >= 1; --i) sa[cnt[rk[i]]--] = i;

  for (w = 1; w < n; w <<= 1, m = p) {
    for (p = 0, i = n; i > n - w; --i) id[++p] = i;
    for (i = 1; i <= n; ++i)
      if (sa[i] > w) id[++p] = sa[i] - w;
    memset(cnt, 0, sizeof(cnt));
    for (i = 1; i <= n; ++i) ++cnt[key1[i] = rk[id[i]]];
    for (i = 1; i <= m; ++i) cnt[i] += cnt[i - 1];
    for (i = n; i >= 1; --i) sa[cnt[key1[i]]--] = id[i];
    memcpy(oldrk + 1, rk + 1, n * sizeof(int));
    for (p = 0, i = 1; i <= n; ++i)
      rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
  }

  while (l <= r) {
    printf("%c", rk[l] < rk[n + 1 - r] ? s[l++] : s[r--]);
    if ((++tot) % 80 == 0) puts("");
  }

  return 0;
}
```